Programación Distribuida – Cloud Haskell

Programación Funcional Avanzada

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Programación Distribuida

If you book them, they will come

- Ejecutar programas simultáneamente en varias máquinas.
- Uso más eficiente de los recursos de red – desplegar servidores cerca de los clientes.
- Explotar ambientes heterogéneos.
- `forkIO`, `MVar`, `STM`, `TVar` no son suficientes
  - ¿Cómo gestionar fallas parciales de hardware?
  - ¿Cómo regular el costo de compartir datos?
  - ¿Cómo garantizar consistencia en el estado?

Modelo de Actor
Pasaje de Mensajes

UNIVERSIDAD SIMÓN BOLÍVAR

Infraestructura de desarrollo

Una familia de librerías

- Programación distribuida construida con librerías que aprovechan las facilidades propias de Haskell.
- API central – `distributed-process`
- Capa de transporte
  - Componente de comunicaciones para intercambio de mensajes.
  - Depende del modelo físico de red – local, TCP, Infiniband...
  - Arranque, inicio y descubrimiento de nodos.
- Librería reciente y con mucha actividad de desarrollo – suficientes características para hacerla práctica y viable.

UNIVERSIDAD SIMÓN BOLÍVAR

Infraestructura principal

`Control.Distributed.Process`

- Iniciar procesos local o remotamente.
- Serializar datos Haskell para intercambiar mensajes.
- Vincular procesos (*process linking*) para recibir notificaciones de falla.
- Recepción de mensajes por múltiples canales.
- Canales dedicados para recibir mensajes con tipo dinámico.
- Detección e identificación automática de nodos.

UNIVERSIDAD SIMÓN BOLÍVAR

Procesos y Nodos

Conceptos básicos

- Un programa consiste de una colección de procesos, cada uno con un identificador único, posiblemente en varios nodos

```
data Process     -- Monad, MonadIO
data NodeId      -- Eq, Ord, Show, Typeable, Binary
data ProcessId   -- Eq, Ord, Show, Typeable, Binary
```

- Un proceso puede saber su nodo e identificador.

```
getSelfNode :: Process NodeId
getSelfPid  :: Process ProcessId
```

- Pueden iniciarse procesos pasando una clausura

```
spawn :: NodeId -> Closure (Process ())
      -> Process ProcessId
```

- Procesos corren en el Monad `Process` – hacer I/O vía `liftIO`.

UNIVERSIDAD SIMÓN BOLÍVAR

Mensajes

Conceptos básicos

- Se envían mensajes a un proceso específico – asíncronamente.

```
send :: Serializable a
     => ProcessID -> a -> Process ()
```

- Un proceso puede esperar mensajes – síncronamente.

```
expect :: Serializable a
       => Process a
```

- Cualquier tipo de datos que sea instancia de `Serializable`
  - Basta derivar `Binary` y `Typeable`
  - La instancia de `Binary` puede especificarse.
  - GHC es capaz de derivar `Binary` a partir de `Generic`.

UNIVERSIDAD SIMÓN BOLÍVAR

Un nodo, un proceso y un mensaje

Ejemplo trivial de la maquinaria

```
main :: IO ()
main = do
  Right t <- createTransport "127.0.0.1"
                             "10501"
                             defaultTCPParameters
  node <- newLocalNode t initRemoteTable
  _ <- forkProcess node $ do
    self <- getSelfPid
    send self "hello"
    hello <- expect :: Process String
    liftIO $ putStrLn hello
  return ()
```

- `createTransport` – prepara la comunicación TCP.
- `newLocalNode` – nodo multiproceso asociado al transporte.
- `forkProcess` – iniciar proceso en el nodo.

UNIVERSIDAD SIMÓN BOLÍVAR

Un nodo, dos procesos, un mensaje

. . . y *logging* al costado

```
replyBack :: (ProcessId, String) -> Process ()
replyBack (sender, msg) = send sender msg

logMessage :: String -> Process ()
logMessage msg = say $ "handling " ++ msg
```

- Si el mensaje recibido es una tupla `(ProcessID,String)`, el proceso reaccionará haciendo eco al remitente.
- Si el mensaje recibido es un `String`, se registrará en el canal de error estándar (*stderr*).

```
say :: String -> Process ()
```

  - `say` envía sus mensajes al proceso `logger`.
  - Todo nodo tiene un `logger` implícito o explícito – el implícito simplemente muestra los mensajes en *stderr*

UNIVERSIDAD SIMÓN BOLÍVAR

Un nodo, dos procesos, un mensaje

. . . y *logging* al costado

```
main = do
  Right t <- createTransport "127.0.0.1" "10501" default
  node <- newLocalNode t initRemoteTable
  forkProcess node $ do
    echoPid <- spawnLocal $ forever $ do
      receiveWait [match logMessage, match replyBack]
    say "send some messages!"
    send echoPid "hello"
    self <- getSelfPid
    send echoPid (self, "hello")
    m <- expectTimeout 1000000
    case m of
      Nothing  -> die "nothing came back!"
      (Just s) -> say $ "got " ++ s ++ " back!"
    return ()
  liftIO $ threadDelay (1*1000000)
  return ()
```

Un nodo, dos procesos, un mensaje

. . . y maquinaria interesante

- Cualquier proceso puede iniciar tantos otros como necesita – en nuestro ejemplo, el principal usa `spawnLocal`.
- El proceso "eco" acepta entre varios mensajes posibles

```
receiveMatch :: [Match b] -> Process b
match :: (Typeable a, Binary a)
      => (a -> Process b) -> Match b
```

- El proceso principal envía mensajes de diferentes tipos – un `String` y un `(ProcessId,String)`.
- `expectTimeout` permite poner una cota al tiempo de espera por mensajes para no bloquear indefinidamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Topologías

Detección automática de nodos

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    ["master", host, port] -> do
      backend <- initializeBackend host port
                                   initRemoteTable
      startMaster backend (const (loop backend))
    ["slave", host, port] -> do
      backend <- initializeBackend host port
                                   initRemoteTable
      startSlave backend
```

- Argumentos establecen rol y parámetros
- Una vez conectados al *backend*, desplegamos.

UNIVERSIDAD SIMÓN BOLÍVAR

Despliegue y uso

Gracias al *backend*

- Despliegue provisto por `localnet`

```
startMaster :: Backend
            -> ([NodeId] -> Process ())
            -> IO ()
startSlave :: Backend -> IO ()
```

- Si hay varios clientes previamente iniciados, el *backend* los detecta y notifica al maestro.
- El maestro puede detectarlos o detenerlos

```
findSlaves          :: Backend -> Process [ProcessId]
terminateSlave      :: NodeId -> Process ()
terminateAllSlaves :: Backend -> Process ()
```

UNIVERSIDAD SIMÓN BOLÍVAR

Topologías

Control de natalidad

```
loop :: Backend -> Process ()
loop backend = do
  liftIO $ threadDelay $ 1000 * 1000
  pids <- findSlaves backend
  say $ "\n\tslaves:\n"  ++ (unlines $ map show pids)
  if (length pids == 4)
     then terminateAllSlaves backend
     else loop backend
```

- Logging de los nodos esclavos se redirigen al maestro – usar `redirectLogHere` si aparecen nuevos esclavos.
- Los nodos esclavos no hacen nada – intencional en el ejemplo.
- Usar `spawn` para iniciar procesos en los esclavos.

UNIVERSIDAD SIMÓN BOLÍVAR

Enviar y recibir

A tener en cuenta

- `send` – enviar mensajes
  - Asíncrono – el enviador **no** se bloqueará.
  - **Nunca** falla – el mensaje se envía, ojalá se reciba.
  - No hay garantía en cuanto al tiempo de entrega.
  - No hay garantía de recepción.
  - Los mensajes se acumulan en orden de llegada al proceso.
- `expect` – aceptar mensajes específicos.
  - Síncrono – el receptor se bloqueará.
  - Otros mensajes son recibidos pero no aceptados.
- `receive` o `matchAny` – filtrar mensajes.

UNIVERSIDAD SIMÓN BOLÍVAR

Controlando la recepción de mensajes

- Procesar los mensajes en *estricto* orden de llegada requiere una función auxiliar

```
matchAny :: (Message -> Process a) -> Match a
```

- Es posibe reenviar todos los mensajes *sin* decodificar

```
forward :: Message -> ProcessId -> Process ()
```

- Decidir si un mensaje es interesante o no

```
handleMessage :: (Monad m, Serializable a)
                 => Message
                 -> (a -> m b) -> m (Maybe b)
```

  - `Nothing` – indica un mensaje indescifrable
  - `Just x` – `x` es la razón de interés.

UNIVERSIDAD SIMÓN BOLÍVAR

Proxy de mensajes

```
proxy :: ProcessId -> (a -> Process Bool) -> Process ()
proxy pid proc = do
  receiveWait [
    matchAny (\m -> do
      next <- handleMessage m proc
      case next of
        Just True  -> forward m pid
        Just False -> return ()
        Nothing    -> return ())
    ]
  proxy pid proc
```

- `proc` produce un `Naybe Bool`
- Reenvía a `pid` aquellos mensajes "interesantes" según el criterio `proc`.

UNIVERSIDAD SIMÓN BOLÍVAR

Vida de los Procesos

They spawn until they die...

- Un proceso se mantiene en ejecución hasta que completa su evaluación o es interrumpido.
- Las excepciones no manejadas los interrumpen de manera abrupta.
- `exit` – envía la excepción `ProcessExitException`.
- `kill` – envía la excepción `ProcessKillException`.
- `die` – facilita el suicido de un proceso.

¿Y si es un proceso "importante"?

UNIVERSIDAD SIMÓN BOLÍVAR

Enlaces (*Links*)

Dependencias entre procesos

```
link     :: ProcessId -> Process ()
linkNode :: NodeId -> Process ()
```

- Un proceso puede enlazarse con otros procesos o nodos.
- Los enlaces son *unidireccionales*.
- Si un proceso A está enlazado con un objeto B, y el objeto B termina (normal o anormalmente), A también.
- La notificación es una excepción asíncrona *no atrapable*.

UNIVERSIDAD SIMÓN BOLÍVAR

Supervisores (*Monitor*)

Dependencias entre procesos

```
monitor     :: ProcessId -> Process MonitorRef
monitorNode :: NodeId -> Process MonitorRef
```

- Un proceso puede supervisar a otro proceso o nodo.
- Si un proceso A está supervisando un objeto B, y el objeto B termina (normal o anormalmente), A recibe un mensaje.

```
data ProcessMonitorNotification =
  ProcessMonitorNotification
    !MonitorRef !ProcessId !DiedReason

data NodeMonitorNotification =
  ProcessMonitorNotification
    !MonitorRef !NodeId !DiedReason
```

UNIVERSIDAD SIMÓN BOLÍVAR

Canales con Tipo Fijo

Cuando todos los mensajes son iguales

```
newChan      :: (Typeable a, Binary a)
             => Process (SendPort a, ReceivePort a)
sendChan     :: (Typeable a, Binary a)
             => SendPort a -> a -> Process ()
receiveChan  :: (Typeable a, Binary a)
             => ReceivePort a -> Process a
```

- Análogos a los `Chan` o `TChan`
  - Entre procesos arbitrarios en nodos arbitrarios.
  - Diferenciación entre el extremo para escribir y el extremo para leer.
- Bidireccional – útil para mensajes que *deben* recibir respuesta.

UNIVERSIDAD SIMÓN BOLÍVAR

Supervisores Genéricos

Burocracia distribuida

- Un `Supervisor` gestiona una colección de procesos.
- Permiten estructurar las aplicaciones como subsistemas independientes con condiciones específicas de arranque, detención, o reacción a fallas.
- Cada supervisor tiene un árbol de procesos
  - Algunos regulares (*workers*) y otros supervisores.
  - Para cada proceso, se especifica la forma de arranque, la forma de detención y cuándo reiniciarlo.

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Towards Haskell in the Cloud – Epstein, Black, Peyton-Jones
- Documentación de `Control.Distributed.Process`
- Documentación de `Control.Distributed.Process.Backend.SimpleLocalnet`
- Documentación de `Control.Distributed.Process.Supervisor`
- Cloud Haskell en Haskell WiKi

UNIVERSIDAD SIMÓN BOLÍVAR